Static Analysis in Compilers

František Nečas (xnecas27), David Chocholatý (xchoch08)

With software becoming increasingly more complex, a lot of effort is being invested into making it more robust in all stages of the development cycle. While previously, code analysers were able to only find common "code smells"[1], with improvements being done to state-of-the-art static analysers, current tools can reliably detect a multitude of faults introduced by developers. Such tools usually function on their own and are manually or automatically run on the source code during development. However, there have recently been attempts to incorporate some types of analyses into compilers[2]. In this presentation, we are going to give an overview of the current capabilities of static analysis used in compilers, mainly studying the case of *gcc*.

Various forms of code analysis are used across the whole compiler, starting from the semantic analysis stage and ending in the optimization stage. Even if the code being compiled is valid according to the language specification, the compiler may issue warnings for error-prone code fragments. These can be seen as a primitive form of code analysis. We are going to briefly explore their workings, capabilities, and limitations.

We are then going to contrast this to a fully-featured static analysis which can detect further errors thanks to an exploration of various program paths at the cost of taking longer time to compute. Finding bugs using a static analyser in a compiler requires being able to compile the code in a *reasonable* time compared to standalone verification tools which can often take hours or days to run for more complex programs. Therefore, compilers do not try to achieve sound or complete analysis, but rather try to catch the most common bugs which developers may not notice. The main advantage of using static analysis in a compiler is the fact that compilation happens very early in the development process, hence bugs can be caught early in the pipeline. Moreover, the analyser can fully utilize the results (e.g., intermediate representations) of the compiler. While preparing this presentation, we talked to the author of the *fanalyzer* option in *gcc*, David Malcolm, who is a principal software engineer at Red Hat and provided us with details about the analyser's internal workings.

Finally, we are going to mention recent attempts to use static analysis to improve the optimization phase[3], too.

[1] A characteristic of a program which may indicate a problem/fault in the program
[2] An example of this includes *gcc* – https://developers.redhat.com/articles/2022/04/12/state-static-analysis-gcc-12-compiler
[3] As can be seen at https://www.researchgate.net/publication/221148065_Control_Flow_Optimization_in_Loops_using_Interval_Analysis